Obstruction degree: measuring concurrency in shared memory systems

Rachid Guerraoui and Mihai Letia
EPFL
{rachid.guerraoui,mihai.letia}@epfl.ch

Abstract

This paper presents the notion of obstruction degree, a new metric for reasoning about the scalability of concurrent algorithms. Essentially, this simple metric gives an indication as to which of two concurrent algorithms is expected to scale better, as well as approximate the maximum number of processors up to which a concurrent algorithm is expected to scale. The scope of the metric is the arguably large class of concurrent algorithms obtained by augmenting sequential algorithms with locks (pessimistic), invalidation primitives (optimistic) or transactional memory. We illustrate the metric through various algorithms and we convey the accuracy of our scalability predictions through extensive experimental measures.

1 Introduction

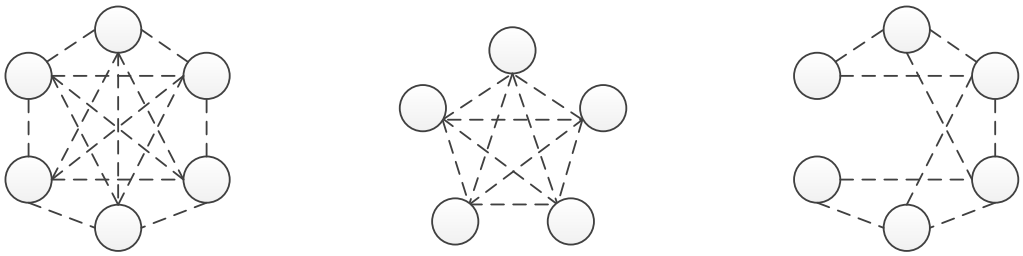
The goal of this work is to characterize the *scalability* of concurrent algorithms a priori, i.e., before implementing them. More specifically, we seek to find a way to compare algorithms based on the performance gain expected as the number of processors increases, as well as predict their *scalability limit*, namely the point after which performance cannot be increased by adding more processors to the system. Predicting scalability is a very challenging problem, and, to the best of our knowledge, finding the point up to which a given algorithm can scale without relying on runtime information, i.e. actually implementing the algorithm, has never been attempted.

We present in this paper a simple approach that contributes to taking up the challenge. The approach is not bulletproof. Yet it gives surprisingly good approximations in many cases and can be viewed as a first step towards determining analytically the scalability of concurrent algorithms.

The starting point of our approach is the simple observation that, in a multicore architecture, the main impediment to scalability is when processes *obstruct* each other. Consider for example the simple task of atomically finding the sum of two variables a and b, and assigning the result to c. For this, a process will first read a, then read b, sum the two, and finally write the result to c. In order to successfully complete this operation, a process must ensure that the values of a and b are not modified until the sum is written to c. Namely, the process *obstructs* others from modifying a and b in order for the three operations appear to execute as a single atomic unit. Intuitively, the less processes obstruct each other, the higher the number of processes that can run concurrently, i.e., the higher the scalability.

The contribution of this paper is to capture this intuition by introducing the notion of *obstruction degree*. In a nutshell, we measure, through a simple integer, the amount of shared data to which an algorithm requires exclusive access, either through locking or invalidation. Intuitively, an operation has obstruction degree k if it is required to (appear to) execute in the same atomic unit as k other operations that were previously executed by the same process. The obstruction degree of an algorithm is then defined as the sum of the obstruction degrees of all the operations it executes. The lower the obstruction degree, the better the algorithm is expected to scale.

As a simple example, consider the application logic behind a computer game: a system consisting of 50 objects and some algorithms that invoke operations provided by these objects. Assume the programmers first design an algorithm that needs to execute an operation on each of 6 different objects, requiring all operations to appear to execute as a single atomic unit. This algorithm, which we denote by A, has obstruction degree 15, as described in Section 3. Not being satisfied with this algorithm, the programmers continue to improve it and end up with two alternative algorithms, B and C:



(a) obstruction degree 15. (b) obstruction degree 10. (c) obstruction degree 9.

Figure 1: Algorithms with different obstruction degrees.

- B executes an operation on each of 5 different objects, where all operations must appear to execute as a single atomic unit;
- C executes an operation on each of 6 different objects, but where each operation is only required to appear to execute in the same atomic unit as the 2 previous operations executed by the algorithm.

Intuitively, both B and C are more scalable than A: B needs to access fewer objects than A, while C accesses the same number of objects as A, but does not require all the operations to appear in the same atomic unit. But which of B and C scales better?

Comparing B and C based on intuition alone is difficult. Our metric reveals useful in this regard. We compute (in Section 3) the obstruction degree to be 10 for B and 9 for C: the latter hence scales better. Figure 1 is a graphical representation of the three algorithms, with circles representing objects and dashed lines linking objects whose operations are required to appear to execute in the same atomic unit. Basically, determining the obstruction degree boils down to counting the number of lines connecting the objects.

After having chosen algorithm C, one might want to determine its scalability limit, i.e., the maximum number of processors up to which this algorithm can scale. Using our metric in a fashion similar to that of the birthday paradox [23], we determine the point where the number of processes running the algorithm without encountering a conflict is maximized. This represents the saturation point when we should not increase concurrency any further. For instance, running this algorithm on 10 processors would result in 2.4 conflicts on average. One can then find that the scalability limit of Algorithm C is 32 processors, that of B is 24 and of A is 19.

We compute the obstruction degree of several well-known algorithms in the paper and explore their scalability limit. We also show how to classify concurrent algorithms based on the growth of their obstruction degree as a function of the number of objects they access. We define two such classes, those with quadratic and those with linear obstruction degree. We show that simple techniques, such as coarse-grained locking and two-phase locking, induce a quadratic obstruction degree, while other, more involved, techniques such as hand-over-hand locking induce only a linear obstruction degree.

We provide extensive experimental evaluations conveying the accuracy of predictions based on the obstruction degree. We first present a synthetic benchmark through which we show that even a small difference between the obstruction degree of two algorithms has an impact on scalability. We then consider a linked list micro-benchmark as well as the STAMP [25] and STMBench7 [14] transactional memory benchmarks. Finally we analyze Domination Locking [13] using our new metric.

Needless to say, we do not claim the obstruction degree to be a holy grail that precisely characterizes the scalability of every concurrent program through a single number. Our metric was designed for algorithms that represent concurrent versions of sequential algorithms, obtained by using synchronization mechanisms such as locks, transactional memory [18] or universal constructions [17]. The type of optimistic locking obtained by revalidating past reads also falls under the scope of our metric. This technique has been widely used in concurrent programming for implementing both transactional memories [9, 26] and other objects such as the snapshot [19].

Intuitively, obstruction is used to prevent certain histories, considered incorrect by the algorithm designer, from occurring. The obstruction degree represents a simple metric for estimating the number of histories that the synchronization mechanism used by such (initially sequential) algorithms allows to occur. Algorithms that have been built from the ground up to be concurrent by using hardware

synchronization primitives such as compare-and-swap do not use obstruction directly, making it hard to apply our metric for this type of synchronization. A linked list algorithm using compare-and-swap to link and unlink nodes from the list is such an example where obstruction is not used directly.

Even within the scope of concurrent algorithms that we consider, the notion of obstruction degree has its limitations, which we discuss further in the paper. For instance, we show that the most accurate predictions are obtained when each operation of each object has the same probability of being invoked. However, even in situations where this condition is not fully satisfied, we show our metric to make relatively good predictions.

Roadmap After recalling our general concurrency model in Section 2, we describe our notion of obstruction degree in Section 3. We present our new metric in an abstract, hopefully easy to understand way, before delving into details about how to use the obstruction degree for obtaining more precise predictions under various scenarios. We compute in Section 4 the obstruction degree of several well-known algorithms and we explore the limits of scalability using our new metric in Section 5. We provide experimental results in Section 6 and we show how the obstruction degree can be used together with the classical notion of step complexity in Section 7. Finally, we review related work in Section 8 and discuss the limitations of our approach in Section 9.

2 Notations and definitions

We introduce our notion of obstruction degree in a standard model of a shared-memory system [5], consisting of several sequential threads of control called *processes* and shared data structures called *objects*. Objects represent the state of the system: they provide operations through which processes examine and change the system state. In order to avoid confusion with the multi-object operations we consider in the rest of the paper, we refer to these single-object operations as *primitives* and we say that a process *applies* a primitive to an object. We denote by $\mathcal{O}$ the set of objects in the system and, for any object $o \in \mathcal{O}$, $o.prim$ denotes the set of primitives provided by o and $o.vals$ is the set of response values to these primitives.

A *protection element* $\epsilon(o)$ is an abstract notion associated to each object o. Each object o has associated a unique protection element $\epsilon(o)$. We model concurrency control by having processes acquire and release protection elements. After process p acquires $\epsilon(o)$ and before it releases it, we say that $\epsilon(o)$ belongs to the *protected set* of p. Before applying a primitive to an object o, process p must acquire the corresponding protection element $\epsilon(o)$. We allow a protection element to belong to the protection set of only one process at any point in time, hence it is required for process p to release $\epsilon(o)$ before another process p' can acquire it.

Note that acquiring or releasing a protection element does not necessarily imply a write to shared memory. For example, an invisible read of a transactional memory is said to acquire the respective memory location to the transaction's protected set, but instead of writing some metadata, the thread having performed the acquisition periodically rechecks the location for consistency. If a concurrent process has updated the location, the initial transaction must abort. We hence allow a process to release a protection element without actively performing a step.

To model the behavior of the kind of system described above, we focus more specifically on three types of events:

- process p applies primitive pr on object o and receives response value v, denoted by $\langle o, pr, v, p\rangle$;
- process p acquires protection element $\epsilon(o)$, denoted by $\langle a(\epsilon(o)), p\rangle$;
- process p releases protection element $\epsilon(o)$, denoted by $\langle r(\epsilon(o)), p\rangle$.

We consider an execution of the system to be a sequence of such events. We use the $\cdot$ operator to denote the concatenation of event sequences and $\mathcal{E}$ is the empty sequence. For a sequence H and a process p, we denote by $H|p$ the subsequence of H containing all the events involving process p. Similarly, for an object o, $H|o$ is the subsequence of H containing all primitive application events on object o, as well as all acquisition and release events of protection element $\epsilon(o)$. For a sequence of events H, we denote by $prot(H)$ the subsequence of H containing all the acquisition and release events. Not all event sequences make sense as executions of the system. Hence we restrict our reasoning to sequences H satisfying the following conditions:

- For every object o, the sequence $prot(H|o)$ must consist of an alternating sequence of acquire and release events, starting with an acquire event. Any acquire event and the immediately following release event in $prot(H|o)$ must involve the same process.
- For every object o, every primitive application event $\langle o, pr, v, p\rangle$ in sequence $H|o$ must be between an acquire event involving the same process $\langle a(\epsilon(o)), p\rangle$ and the immediately following release event involving process p, $\langle a(\epsilon(o)), p\rangle$.

In this work we are concerned only with such well-formed sequences that we refer to as *histories*.

For a history H, a *process subhistory* $H^p = H|p$ is said to be *complete* if for every object o that appears in H^p, the sequence $H^p|o$ ends with a release event $\langle r(\epsilon(o)), p\rangle$. History H is said to be a *single-process history* if all the events in H involve the same process. A complete single-process history is called an *operation* and we say process p executes the operation. For an operation H^p we define its *protection operation* as $prot(H^p)$.

To illustrate, for operation $H^p = \langle a(\epsilon(o_1)), p\rangle, \langle o_1, op_1, v_1, p\rangle, \langle a(\epsilon(o_2)), p\rangle, \langle o_2, op_2, v_2, p\rangle, \langle r(\epsilon(o_1)), p\rangle, \langle r(\epsilon(o_2)), p\rangle$ we obtain its protection operation to be $prot(H^p) = \langle a(\epsilon(o_1)), p\rangle, \langle a(\epsilon(o_2)), p\rangle, \langle r(\epsilon(o_1)), p\rangle, \langle r(\epsilon(o_2)), p\rangle$.

Histories represent the classical way of reasoning about algorithms in distributed computing [4, 5, 20] and they can be inferred from the algorithm pseudocode. As we detail in Section 3, we only need to consider single-process histories when computing the obstruction degree, making our metric easier to compute than other existing solutions [1–3].

3 The obstruction degree

We now define our notion of obstruction degree after describing the design goals we used for our metric. We first give a simple definition of our new metric, in an abstract context, and then continue to describe the extensions we use for comparing algorithms using commutative operations, different locking granularities or revalidation.

3.1 Design goals

Our main objective when designing the obstruction degree was simplicity. We aimed for our metric to be a simple way of comparing concurrent algorithms and approximating the limit up to which they can scale. We argue that a method requiring a complex analysis would be an effort comparable to implementing the algorithm and testing it on actual hardware.

Single-process analysis is our main pathway to simplicity. Programmers can get information about the concurrency of their algorithms without needing concurrent reasoning. Considering all possible interleavings and all intermediate states of a concurrent program is notoriously difficult. However, by not using information about operations being executed by other processes, we do not have perfect information about possible conflicts but estimate it instead.

Hardware independence comes at a cost in terms of precision, but allows our metric to capture the inherent scalability of a concurrent algorithm. We hence do not take into account the cost of synchronization, but measure the amount of synchronization taking place instead. We thus obtain a theoretical metric that nonetheless has practical implications, being accurate enough to be useful for practical comparisons.

3.2 Abstract definition

Given a single-process history H, we define the *protected set* of H, denoted by $P(H)$, as the set of protection elements $\epsilon(o)$ such that H contains an acquisition event $\langle a(\epsilon(o)), p\rangle$ that is not followed by a matching release event. We denote the cardinal of $P(H)$ by $|P(H)|$.

For simplicity of presentation, we assume that any two distinct objects are protected by distinct protection elements. That is, $\forall o, o' \in \mathcal{O}$, we have $\epsilon(o) \neq \epsilon(o')$. In Section 3.5 we describe how we determine the obstruction degree for algorithms that do not satisfy this property, such as coarse-grained locking algorithms.

Definition 3.1 *Given a protection operation H, we define the obstruction degree $C(H)$ as follows:*

- ***OD1:*** *if $H = \mathcal{E}$, then $C(H) = 0$;*

- ***OD2:*** *if* $H = H' \cdot \langle a(\epsilon(o)), p\rangle$*, then* $C(H) = C(H') + |P(H')|$*;*
- ***OD3:*** *if* $H = H' \cdot \langle r(\epsilon(o)), p\rangle$*, then* $C(H) = C(H')$*.*

Informally, whenever an operation acquires a new protection element, its complexity increases by the current size of its protected set. Note that we do not measure through the obstruction degree the amount of time an operation needs to wait in order to acquire the needed protection elements but the potential amount of time it causes other operations to wait. In other words, the obstruction degree measures the obstruction that an operation causes to other operations in the system.

To illustrate how we calculate the obstruction degree, let us go back to the three algorithms, A, B, C, presented in the introduction. The executions of these algorithms are operations ω_A, ω_B and ω_C that have protection operations $prot(\omega_A)$, $prot(\omega_B)$ and $prot(\omega_C)$.

Since operation ω_A needs to atomically apply primitives to six objects, it needs to acquire the six protection elements, apply the primitives and finally release the protection elements. Hence the protection operation $prot(\omega_A)$ is $prot(\omega_A) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle a(\epsilon_4), p\rangle, \langle a(\epsilon_5), p\rangle, \langle a(\epsilon_6), p\rangle, \langle r(\epsilon_1), p\rangle, \langle r(\epsilon_2), p\rangle, \langle r(\epsilon_3), p\rangle, \langle r(\epsilon_4), p\rangle, \langle r(\epsilon_5), p\rangle, \langle r(\epsilon_6), p\rangle$. By applying rule **OD3** 6 times we obtain that $C(prot(\omega_A)) = C(prot(\omega'_A))$, where $prot(\omega'_A) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle a(\epsilon_4), p\rangle, \langle a(\epsilon_5), p\rangle, \langle a(\epsilon_6), p\rangle$. Next we apply rule **OD2** 5 times and we obtain that $C(prot(\omega_A)) = C(\langle a(\epsilon_1)\rangle) + 5 + 4 + 3 + 2 + 1 = 15$.

Operation ω_B is similar to ω_A, but only acquires five protection elements instead of six. We then have $prot(\omega_B) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle a(\epsilon_4), p\rangle, \langle a(\epsilon_5), p\rangle, \langle r(\epsilon_1), p\rangle, \langle r(\epsilon_2), p\rangle, \langle r(\epsilon_3), p\rangle, \langle r(\epsilon_4), p\rangle, \langle r(\epsilon_5), p\rangle$. Using the same technique, we obtain that $C(prot(\omega_B)) = C(\langle a(\epsilon_1)\rangle) + 4 + 3 + 2 + 1 = 10$.

Algorithm C requires each primitive to appear to be applied in the same atomic unit as the 2 previous primitives applied, hence each protection element is released after the next 2 have been acquired. Then the protection operation $prot(\omega_C)$ is $prot(\omega_C) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle r(\epsilon_1), p\rangle, \langle a(\epsilon_4), p\rangle, \langle r(\epsilon_2), p\rangle, \langle a(\epsilon_5), p\rangle, \langle r(\epsilon_3), p\rangle, \langle a(\epsilon_6), p\rangle, \langle r(\epsilon_4), p\rangle, \langle r(\epsilon_5), p\rangle, \langle r(\epsilon_6), p\rangle$. We first apply rule **OD3** 3 times and we obtain that $C(prot(\omega_C)) = C(prot(\omega_C^{(1)}))$, where $prot(\omega_C^{(1)}) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle r(\epsilon_1), p\rangle, \langle a(\epsilon_4), p\rangle, \langle r(\epsilon_2), p\rangle, \langle a(\epsilon_5), p\rangle, \langle r(\epsilon_3), p\rangle, \langle a(\epsilon_6), p\rangle$. Next we apply rule **OD2** and we obtain that $C(prot(\omega_C)) = C(prot(\omega_C^{(1)})) = C(prot(\omega_C^{(2)})) + |P(prot(\omega_C^{(2)}))|$, where $prot(\omega_C^{(2)}) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle r(\epsilon_1), p\rangle, \langle a(\epsilon_4), p\rangle, \langle r(\epsilon_2), p\rangle, \langle a(\epsilon_5), p\rangle, \langle r(\epsilon_3), p\rangle$ and $|P(prot(\omega_C^2))| = 2$. We apply rule **OD3** once again followed by rule **OD2** and we obtain that $C(prot(\omega_C)) = C(prot(\omega_C^{(3)})) + |P(prot(\omega_C^{(3)}))| + 2$, where $prot(\omega_C^{(3)}) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle r(\epsilon_1), p\rangle, \langle a(\epsilon_4), p\rangle, \langle r(\epsilon_2), p\rangle$ and $|P(prot(\omega_C^{(3)}))| = 2$. We again apply rules **OD3** and **OD2** and we obtain that $C(prot(\omega_C)) = C(prot(\omega_C^{(4)})) + |P(prot(\omega_C^{(4)}))| + 4$, where $prot(\omega_C^{(4)}) = \langle a(\epsilon_1), p\rangle, \langle a(\epsilon_2), p\rangle, \langle a(\epsilon_3), p\rangle, \langle r(\epsilon_1), p\rangle$ and $|P(prot(\omega_C^{(4)}))| = 2$. In the end we apply rule **OD3** once and rule **OD2** 3 times and we obtain that $C(prot(\omega_C)) = C(\mathcal{E}) + 1 + 2 + 6 = 9$.

Although our scalability metric can be computed for any shared memory algorithm, the accuracy of its predictions are based on the assumption that object implementations are disjoint-access parallel [22]. That is, for any two objects $o, o' \in \mathcal{O}$, any primitives $pr \in o.prim$ and $pr' \in o'.prim$ must be disjoint-access parallel. This ensures that primitives pr and pr' accessing different objects do not interfere and delay each other, thus further affecting scalability. If object implementations are not disjoint-access parallel, one can consider the lower-level primitives applied by pr and pr' along with the protection elements they acquire when reasoning about the obstruction degree of the algorithm, thus obtaining more accurate predictions regarding scalability.

3.3 Commutative accesses

In order to increase concurrency, many systems distinguish read-only from update primitives. The update primitive is typically more expensive than the read-only one, as the latter does not need to obstruct other concurrent read-only primitives.

One approach to making this distinction are read-write locks. If an operation acquires the lock in read mode, other operations can acquire the same lock in read mode, while no operation can acquire it in write mode. If however, an operation acquires the lock in write mode, no other operation can acquire it in either read or write mode.

To reason about the synchronization mechanism of a system that allows read-only primitives to be applied concurrently, we extend our model as follows. To each object $o \in \mathcal{O}$, we associate an array $\epsilon(o)$ of protection elements of size n, where n is the number of concurrent processes running in the system. When process p needs to apply a read-only primitive on object o, it acquires protection element $\epsilon(o)[p]$, whereas if the process needs to apply an update primitive, it acquires all protection elements in $\epsilon(o)$, namely $\epsilon(o)[1], \epsilon(o)[2], \ldots, \epsilon(o)[n]$. This approach allows distinct processes to acquire distinct protection elements and therefore apply read-only primitives concurrently. Before applying an update primitive, a process needs to acquire all protection elements and therefore does not allow for any other process to concurrently apply a primitive. This way we consider that applying an update primitive is n times more expensive than for applying a read-only one.

Systems where other, more complex, commutative [1] primitives can be applied concurrently are modeled in a similar way. Instead of associating a protection element $\epsilon(o)$ with each object $o \in \mathcal{O}$, we consider an array $\epsilon(o, pr)$ of size n of protection elements associated with each primitive $pr \in o.prim$. Before process p can apply primitive pr to object o, it must acquire all protection elements from all the arrays $\epsilon(o, pr')$ associated with all primitives $pr' \in o.prim$ that do not commute with pr as well as the protection element $\epsilon(o, pr)[p]$. This ensures that all primitives that can be applied concurrently acquire disjoint sets of protection elements and that all primitives that cannot must acquire intersecting sets of protection elements.

3.4 Beyond locking

The notion of protection element does not apply only to locking, but to lock-free synchronization as well. Consider Algorithms 1 and 2 below presenting two different ways of atomically reading the values of two distinct memory locations.

Algorithm 1 Lock-based double atomic read

```
 1: read-atomic-lock(x,y):
 2:   loop
 3:     if L_x.lock() then
 4:       if L_y.lock() then
 5:         x' ← x.read()
 6:         y' ← y.read()
 7:         L_x.unlock()
 8:         L_y.unlock()
 9:         return(x', y')
10:       else
11:         L_x.unlock()
12:   end loop
```

Algorithm 1 conveys a locking implementation. The process attempts to acquire the two locks corresponding to the memory locations, upon success it reads the two locations and finally releases the locks and returns the values. Executing this algorithm would result in the following operation being executed by a process p: $\omega_1 = \langle a(L_x), p\rangle, \langle a(L_y), p\rangle, \langle x, read, v_x, p\rangle, \langle y, read, v_y, p\rangle, \langle r(L_x), p\rangle, \langle r(L_y), p\rangle$. The protection operation of ω_1 is then $prot(\omega_1) = \langle a(L_x), p\rangle, \langle a(L_y), p\rangle, \langle r(L_x), p\rangle, \langle r(L_y), p\rangle$ which according to Definition 3.1 has obstruction degree 1.

Algorithm 2 Revalidation-based double atomic read

```
 1: read-atomic-nolock(x,y):
 2:   loop
 3:     x' ← x.read()
 4:     y' ← y.read()
 5:     if revalidate(x,x') then return(x', y')
 6:   end loop

 7: revalidate(x,x'):
 8:   if x' = x.read() then
 9:     return true
10:   else
11:     return false
```

[1] Any two read-only operations commute while a read and an updated do not generally commute.

Although using a different synchronization mechanism, Algorithm 2 serves the same purpose, namely obtaining two values that were in memory at the same point in time. In order to achieve this, the process reads the two memory locations and revalidates the first, then upon success it returns the two values. Executing this algorithm would result in the following operation being executed by a process p: $\omega_2 = \langle a(\epsilon(x)), p\rangle, \langle x, read, v_x, p\rangle, \langle a(\epsilon(y)), p\rangle, \langle y, read, v_y, p\rangle, \langle r(\epsilon(y)), p\rangle, \langle r(\epsilon(x)), p\rangle$. Similar to ω_1, this operation has the protection operation $prot(\omega_2) = \langle a(\epsilon(x)), p\rangle, \langle a(\epsilon(y)), p\rangle, \langle r(\epsilon(y)), p\rangle, \langle r(\epsilon(x)), p\rangle$ which according to Definition 3.1 also has obstruction degree 1.

The two algorithms described above, although using different synchronization mechanisms, have the same obstruction degree. This obstruction is inherent to the desired semantics of the operation since the process executing it must be able to read the second value without the first one being modified by a concurrent process. In Section 6.1 we compare the performance of two slightly more complex variants of these algorithms and we show that they provide very similar scalability.

3.5 Different granularities

So far we assumed that for any two objects $o, o' \in \mathcal{O}$, the corresponding protection elements are different, $\epsilon(o) \neq \epsilon(o')$. However, for some synchronization mechanisms, most notably coarse-grained locking, this property does not hold. Indeed, in the case of coarse-grained locking $\forall o, o' \in \mathcal{O}$ we have $\epsilon(o) = \epsilon(o')$.

Since process p is required to acquire protection element $\epsilon(o)$ before accessing object o, another process p' cannot concurrently access object $o' \neq o$ if $\epsilon(o) = \epsilon(o')$. Given an object $o \in \mathcal{O}$, we define the *set of neighbors* of o as $N(o) = \{o' \in \mathcal{O} | \epsilon(o) = \epsilon(o')\}$. A system for which $\exists o \in \mathcal{O}$ such that $|N(o)| > 1$ is said to contain neighbors. To obtain a value for the obstruction degree that better predicts the scalability of a system that contains neighbors, instead of computing the cost of an operation ω, we compute the cost of operation ω', obtained from ω as follows. We replace each acquisition event $\langle a(\epsilon(o)), p\rangle$ of operation ω with the sequence $\langle a(\epsilon(o)^{(1)}), p\rangle, \ldots, \langle a(\epsilon(o)^{(m)}), p\rangle$ and similarly for the release events, with $m = |N(o)|$.

4 Obstruction degree classes

We define here obstruction degree classes as well as classify well-known algorithm into these classes.

4.1 Quadratic obstruction degree

Definition 4.1 *An algorithm has quadratic obstruction degree in terms of the number of protection elements it acquires m if its execution is operation ω and $C(\omega)$ is in $O(m^2)$.*

Lemma 4.2 *For any two protection elements l_1 and l_2, any operation ω_1 having protection operation $prot(\omega_1) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \langle r(l_2), p\rangle$ has obstruction degree 1.*

Proof. We first apply rule **OD3** twice and we obtain that $C(prot(\omega_1)) = C(prot(\omega_2))$, where $prot(\omega_2) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle$. We now see that $prot(\omega_2) = prot(\omega_3) \cdot \langle a(l_2), p\rangle$, with $prot(\omega_3) = \langle a(l_1), p\rangle$. By applying rule **OD2** we find that $C(prot(\omega_2)) = C(prot(\omega_3)) + |P(prot(\omega_3))|$, where $|P(prot(\omega_3))| = 1$. We use again rule **OD2** and find that $C(prot(\omega_3)) = C(\mathcal{E}) + P(\mathcal{E})$ and by rule **OD1**, $C(prot(\omega_3)) = 0 + 0 = 0$. We obtain that $C(prot(\omega_1)) = C(prot(\omega_2)) = C(prot(\omega_3)) + 1 = 1$ and our proof is complete. □

Lemma 4.3 *For any m protection elements $l_1, \ldots, l_m$, any operation ω_m having protection operation $prot(\omega_m) = \langle a(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle r(l_m), p\rangle$ has obstruction degree $m(m-1)/2$.*

Proof. We conduct this proof by induction.

The base case. The obstruction degree of the protection operation $prot(\omega_1) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \langle r(l_2), p\rangle$ is 1. See Lemma 4.2 for the proof.

The inductive step. We show here that if the obstruction degree of the protection operation $prot(\omega_m) = \langle a(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle r(l_m), p\rangle$ is equal to $m(m-1)/2$, then that of protection operation $prot(\omega_{m+1}) = \langle a(l_1), p\rangle, \ldots, \langle a(l_{m+1}), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle r(l_{m+1}), p\rangle$ is equal to $m(m+1)/2$. By applying rule **OD3** m times to $prot(\omega_m)$ and $m+1$ times to $prot(\omega_{m+1})$, we obtain protection operations $prot(\omega'_m) = \langle a(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle$ and $prot(\omega'_{m+1}) = \langle a(l_1), p\rangle, \ldots, \langle a(l_{m+1}), p\rangle$ such that $C(prot(\omega'_m)) = C(prot(\omega_m))$ and $C(prot(\omega'_{m+1})) = C(prot(\omega_{m+1}))$.

We now have to show that if the obstruction degree of protection operation $prot(\omega'_m)$ is $m(m-1)/2$, then that of ω'_{m+1} is $m(m+1)/2$. We notice that $prot(\omega'_{m+1}) = prot(\omega'_m) \cdot \langle a(l_{m+1}), p\rangle$ and according to rule **OD2**, $C(prot(\omega'_{m+1})) = C(prot(\omega'_m)) + |P(prot(\omega'_m))|$. Since in $prot(\omega'_m)$ there are m protection elements that are acquired and not released, we have $|P(prot(\omega'_m)| = m$. Therefore $C(prot(\omega'_{m+1})) = C(prot(\omega'_m)) + m = m(m-1)/2 + m = m(m+1)/2$ and our proof is complete. □

The two-phase locking protocol [11] is the standard locking protocol for ensuring serializability both in databases and concurrent programs. The name stems from the fact that operations are divided into two phases: the expanding phase, when locks are acquired but not released, and the shrinking phase, when locks are released but not acquired. Although more efficient two-phase locking uses both shared and exclusive locks, we only consider exclusive locks as this allows us to calculate the obstruction degree of operations by considering only their length. The obstruction degree of operations using shared locks depends on the specific workload and can be calculated using the technique described in Section 3.3.

Theorem 4.4 *Any operation that uses the two-phase locking protocol to access m distinct objects has an obstruction degree in $O(m^2)$.*

Proof. Let $x_1, \ldots, x_m$ be the objects that the system protects using locks $l_1, \ldots, l_m$. As operation ω applies primitives to each of $x_1, \ldots, x_m$, according to the two-phase locking protocol, it needs to acquire locks $l_1, \ldots, l_m$. As the protocol requires operations to not release any locks until all the locks have been acquired, the protection operation of ω is then $prot(\omega) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_1), p\rangle, \langle r(l_2), p\rangle, \ldots, \langle r(l_m), p\rangle$. According to Lemma 4.3, the obstruction degree of ω is $m(m-1)/2$ which is in $O(m^2)$. □

To illustrate, consider a linked list with operations insert, remove and search implemented using two-phase locking. An operation that goes up to position k in the list has obstruction degree $k(k-1)/2$. Assuming that, on average, operations go up to the middle of the list, i.e. position $m/2$ out of m, these operations have obstruction degree $m^2/4$, which is in $O(m^2)$.

Coarse-grained locking is a synchronization mechanism where all objects in the system are protected by a single lock. Any process that executes an operation on these objects must acquire this lock. Such a system only allows a single operation to execute at any point in time.

Theorem 4.5 *Any operation that acquires a coarse-grained lock for a system composed of m objects has an obstruction degree in $O(m^2)$.*

Proof. An operation acquiring a coarse-grained lock has the form $\omega = \langle a(\epsilon(o_1)), p\rangle, \langle o_1, op_1, v_1, p\rangle, \ldots, \langle o_k, op_k, v_k, p\rangle, \langle r(\epsilon(o_1)), p\rangle$. If the number of objects in the system is m and $\forall o, o' \in \mathcal{O}, \epsilon(o) = \epsilon(o')$, then the set of neighbors of object o is $N(o) = \mathcal{O}$ and o has $|N(o)| = m$ neighbors. As described in Section 3.5, in order to obtain a correct value of the obstruction degree of ω, we transform ω into operation $\omega' = \langle a(\epsilon(o_1)^{(1)}), p\rangle, \langle a(\epsilon(o_1)^{(2)}), p\rangle, \ldots, \langle a(\epsilon(o_1)^{(m)}), p\rangle, \langle o_1, op_1, v_1, p\rangle, \ldots, \langle o_k, op_k, v_k, p\rangle, \langle r(\epsilon(o_1))^{(1)}, p\rangle, \langle r(\epsilon(o_1))^{(2)}, p\rangle, \ldots, \langle r(\epsilon(o_1))^{(m)}, p\rangle$. We thus obtain protection operation $prot(\omega') = \langle a(\epsilon(o_1)^{(1)}), p\rangle, \langle a(\epsilon(o_1)^{(2)}), p\rangle, \ldots, \langle a(\epsilon(o_1)^{(m)}), p\rangle, \langle r(\epsilon(o_1))^{(1)}, p\rangle, \langle r(\epsilon(o_1))^{(2)}, p\rangle, \ldots, \langle r(\epsilon(o_1))^{(m)}, p\rangle$. Using Lemma 4.3, we find that ω' has obstruction degree $m(m-1)/2$ which is in $O(m^2)$. □

4.2 Linear obstruction degree

Definition 4.6 *An algorithm has linear obstruction degree in terms of the number of protection elements it acquires m if its execution is operation ω and $C(\omega)$ is in $O(m)$.*

Lemma 4.7 *For any m protection elements $l_1, \ldots, l_m$, any operation ω_m having protection operation $prot(\omega_m) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_{m-1}), p\rangle, \langle r(l_m), p\rangle$ has obstruction degree $m-1$.*

Proof. We conduct this proof by induction.

The base case. The obstruction degree of the protection operation $prot(\omega_1) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \langle r(l_2), p\rangle$ is 1. See Lemma 4.2 for the proof.

The inductive step. We continue to show that if the obstruction degree of the protection operation $prot(\omega_m) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_{m-1}), p\rangle, \langle r(l_m), p\rangle$ is $m-1$, then the protection operation $prot(\omega_{m+1}) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_{m+1}), p\rangle, \langle r(l_m), p\rangle, \langle r(l_{m+1}), p\rangle$ has obstruction degree m. By applying rule **OD3** one time to $prot(\omega_m)$ and two times to $prot(\omega_{m+1})$, we obtain $prot(\omega'_m) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_{m-1}), p\rangle$ and $prot(\omega'_{m+1}) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_{m-1}), p\rangle, \langle a(l_{m+1}), p\rangle$ such that $C(prot(\omega'_m)) = C(prot(\omega_m))$ and $C(prot(\omega'_{m+1})) = C(prot(\omega_{m+1}))$.

We now have to show that if the obstruction degree of $prot(\omega'_m)$ is $m-1$, then that of $prot(\omega'_{m+1})$ is m. We notice that $prot(\omega'_{m+1}) = prot(\omega'_m) \cdot \langle a(l_{m+1}), p\rangle$ and using rule **OD2**, we obtain $C(prot(\omega'_{m+1})) = C(prot(\omega'_m)) + |P(prot(\omega'_m))|$. Since in $prot(\omega'_m)$ only l_m is acquired but not released, we have $|P(prot(\omega'_m))| = 1$. Then $C(prot(\omega'_{m+1})) = C(prot(\omega'_m)) + 1 = (m-1) + 1 = m$ and our proof is complete. □

Hand-over-hand locking [6] is a fine grained locking protocol used on data structures such as linked lists and trees. Each node in the data structure is protected by its own lock and every operation starts by acquiring the lock on the head of the list or root of the tree. Then, while still holding that, acquires a lock on the second node; then it releases the first lock and, while still holding the second lock, it acquires the third; and so on. Operations cannot deadlock since locks are always acquired in the same order.

Theorem 4.8 *Any operation that uses the hand-over-hand locking protocol to access m distinct objects has an obstruction degree in $O(m)$.*

Proof. An operation ω using the hand-over-hand locking protocol to access m nodes in a data structure has the form $\omega = \langle a(l_1), p\rangle, \langle o_1, op_1, v_1, p\rangle, \langle a(l_2), p\rangle, \langle o_2, op_2, v_2, p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle o_m, op_m, v_m, p\rangle, \langle r(l_{m-1}), p\rangle, \langle r(l_m), p\rangle$. The protection operation of ω is then $prot(\omega) = \langle a(l_1), p\rangle, \langle a(l_2), p\rangle, \langle r(l_1), p\rangle, \ldots, \langle a(l_m), p\rangle, \langle r(l_{m-1}), p\rangle, \langle r(l_m), p\rangle$. By Lemma 4.7, ω has obstruction degree $m-1$, which is in $O(m)$ and our proof is complete. □

Sedgewick and Flajolet [27] show that the average path length in random binary trees is in $O(n\sqrt{n})$, where n is the number of elements present in the tree. This leads to an average node depth in $O(\sqrt{n})$, hence using the hand-over-hand locking protocol on such a tree has an obstruction degree in $O(\sqrt{n})$. In random binary search trees however, the average path length is in $O(n \log n)$ and the average node depth is in $O(\log n)$. Hence using hand-over-hand locking on a random binary search tree has an obstruction degree in $O(\log n)$.

5 Limits of scalability

The obstruction degree can also help find the limit up to which an algorithm can increase performance while adding more processors. This represents the point when the difference between the number of processes running the algorithm and the ones that encounter conflicts is maximized.

For the reasoning we use to find the scalability limit of an algorithm, we require the extra assumption that all processes are executing the same algorithm, more specifically they are executing operations of the same obstruction degree. We then draw a parallel with the celebrated birthday paradox [23]. The point is to find the minimum number of people that need to be in a room such that there is a certain probability (say 0.5) for two people to share the same birthday. The paradox stems from the fact that although there are 365 days in a year, the number of people that need to be in the room in order to have probability 0.5 is only 23. A specific variant asks the expected number of people that share a birthday given the number of people present in the room. We continue to present a parallel between the number of conflicts in a concurrent program and the number of people sharing a birthday.

A person in the room corresponds to a process. The person has a birthdate, which corresponds to the process executing an operation of a certain obstruction degree, which we denote by O. The operations

have a specified length L, which represents the number of primitives an operation needs to apply. Using the operation length and n, the total number of protection elements in the system, we obtain $L \times n$ which corresponds to the space of dates of birth, the number of days in the year.

We then consider two operations p_A and p_B of obstruction degree O. The probability of operation p_B requiring a protection element which is held by p_A, i.e. generating a conflict, is $\frac{O}{L \times n}$. Then the probability that operation p_B does not require a protection element held by operation p_A is $1 - \frac{O}{L \times n}$. By considering k processes, each executing an operation of obstruction degree O, we obtain the probability that $k-1$ operations do not require a protection element held by operation p_A to be $(1 - \frac{O}{L \times n})^{k-1}$. Similarly, we obtain the expected number of processes that can acquire protection elements which are not held by any other process to be $k \times (1 - \frac{O}{L \times n})^{k-1}$. Finally, the expected number of processes that execute conflicting operations is $c = k \times \left(1 - \left(1 - \frac{O}{L \times n}\right)^{k-1}\right)$.

Definition 5.1 *The* concurrency efficiency $\mathscr{E}$ *of an algorithm is computed as* $\mathscr{E} = 1 - \frac{O}{L \times n}$.

It follows that

$$c = k \times (1 - \mathscr{E}^{k-1}) \tag{1}$$

We obtain the expected number of processes that do not generate a conflict as $r(k) = k - k \times (1 - \mathscr{E}^{k-1}) = k \times \mathscr{E}^{k-1}$. In order to find the maximum of this function we calculate its first derivative to be $r'(k) = \mathscr{E}^{k-1} \times (1 + k \ln \mathscr{E})$, which is zero when $k = -\frac{1}{ln\mathscr{E}}$ or

$$k_{max} = \left\lfloor \log_{1-\frac{O}{Ln}} \frac{1}{e} \right\rfloor \tag{2}$$

When processes cause low obstruction, we have $O \ll Ln$, then $1 - \frac{O}{Ln}$ is close to 1 and we can run a large number of processes, while when O is close to $Ln(1 - \frac{1}{e})$, we have that $1 - \frac{O}{Ln}$ is close to $\frac{1}{e}$, and the number of processes is no more than 1.

Example Consider again Algorithm C described in the introduction, having operations of length $L = 6$ and obstruction degree $O = 9$ in a system of $n = 50$ objects (50 protection elements). Calculating the concurrency efficiency of this algorithm gives $\mathscr{E} = \frac{97}{100}$. Say we first want to find the expected number of conflicts generated when running it on $k = 10$ processors. We use formula (1) and we obtain $c_{10} = k \times (1 - \mathscr{E}^{k-1}) = 2.4$. If we double k and move the same algorithm on a system with $k = 20$ processors, we obtain $c_{20} = 8.8$. Finally, we use formula (2) to find the scalability limit of this algorithm as $k_{max} = 32$. We similarly obtain values of 19 for Algorithm A and 24 for Algorithm B.

As another example, we find the scalability limit of a system where processes execute operations of length L using the two-phase locking protocol. Using Theorem 4.4 we find the obstruction degree of these operations to be $O = L(L-1)/2$. We continue to obtain the concurrency efficiency of the system $\mathscr{E} = 1 - \frac{O}{Ln}$, which gives $\mathscr{E} = 1 - \frac{L-1}{2n}$. We then find the scalability limit as $k_{max} = -\frac{1}{ln\frac{2n-L+1}{2n}}$ and finally $k_{max} = \frac{1}{ln(2n-L+1)-ln(2n)}$.

6 Experimental validation

We use five different benchmarks to evaluate the predictions of our metric. To show the accuracy of the predictions in an environment where each protection element has the same probability of being acquired, we first use a synthetic benchmark where processes read and write shared memory locations uniformly at random. The second is a concurrent linked list micro-benchmark using five of the best-known algorithms [19]. The third is the STMBench7 benchmark, where we compare the two featured locking schemes: coarse and medium grained locking. The fourth is the STAMP benchmark suite, where we consider two benchmarks, namely labyrinth and kmeans. The last is the Domination Locking protocol of Golan-Gueta et al. [13], used for forest-like data structures.

Our test machine has four AMD Opteron 12-core processors running at 2.1GHz, for a total of 48 cores, and has 128GB of memory. In the graphs presented below, each data point is computed by dividing the throughput obtained when using the specified number of threads to the one obtained for a single thread using the same algorithm. The throughput is obtained by averaging over 10 runs of 10 seconds each.

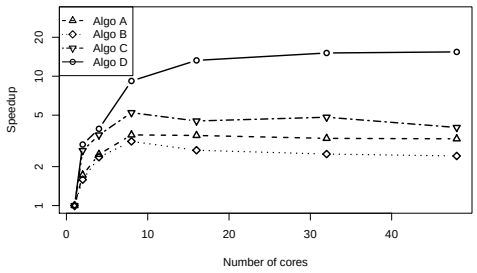


Figure 2: Synthetic benchmark using operations of different obstruction degrees.

The linked list and synthetic benchmark are written in java and we use an extra 5 seconds to warmup the JVM before each run.

6.1 A synthetic benchmark

In order to convey the accuracy of the predictions made using the obstruction degree, we consider a system where any shared object has the same probability of being accessed. We therefore consider operations that randomly read and write the values of a designated number of shared memory locations out of a total of 100. Each memory location is protected by a distinct lock in case of the lock-based algorithms, but we consider lock-free algorithms as well. Small sequential delays that simulate local computation were added to algorithms in order to diminish the impact of the locking overhead on the resulting scalability measurements, as described in Section 9.

We start by comparing four lock-based algorithms. The first, Algorithm A, acquires the locks associated to 8 memory locations before releasing all of them. As an alternative, we consider three more algorithms, all of them acquiring the locks associated to 14 memory locations, but releasing some of them before finishing. As such, we have Algorithm B that does not keep more than 4 locks at any given time, Algorithm C that does not keep more than 3, and Algorithm D that does not keep more than 2 locks. The question that we want to answer is how does Algorithm A compare to Algorithms B, C and D in terms of scalability. This is not an easy comparison as Algorithm A acquires fewer locks, whereas Algorithms B, C and D acquire more locks but keep them for shorter periods of time.

Since Algorithm A acquires 8 locks without releasing any of them until finishing, its obstruction degree is $\sum_{i=1}^{7} i = 28$. Algorithm B acquires 14 locks, but releases one before acquiring the next just after acquiring the fourth lock, having obstruction degree $1 + 2 + 3 * 11 = 36$, greater than that of Algorithm A. Algorithm C starts releasing after acquiring the third lock, thus having obstruction degree $1 + 2 * 12 = 25$, which is smaller than that of Algorithm A. The smallest obstruction degree is that of Algorithm D, namely 13. Computing the concurrency efficiency $\mathscr{E}$ of the four algorithms gives $\mathscr{E}_A = \frac{193}{200}$, $\mathscr{E}_B = \frac{341}{350}$, $\mathscr{E}_C = \frac{55}{56}$ and $\mathscr{E}_D \approx \frac{99}{100}$. We then apply formula (2) to find the maximum number of processes that can efficiently run these algorithms to be $k_{maxA} = 28$, $k_{maxB} = 38$, $k_{maxC} = 55$ and $k_{maxD} = 99$.

Figure 2 shows the measured speedup of the four algorithms. The obstruction degree of Algorithm A lies in between that of Algorithms B and C, and the graph shows that Algorithm A scales better than Algorithm B but not as well as Algorithm C. Algorithm D has a very low obstruction degree, allowing it to scale much better than the other three. Although the obstruction degrees of these algorithms do not differ by a large quantity, the effect can nonetheless be observed in terms of the measured speedup. Algorithms A, B and C scale to only 8 threads and our estimated limit of scalability is at most 55. The limit of scalability of Algorithm D is 99 and it scales up to the 48 cores available on our test machine, while obtaining a very small speedup when moving from 32 to 48 cores.

We continue by comparing two simple algorithms that fulfill the same goal, one using a lock-based approach while the other one is lock-free. Both algorithms attempt to read the values of three memory locations as a single atomic snapshot, followed by a write to another location, which is not required to appear in the same atomic unit as the reads. Both algorithms acquire three protection elements, associated to the three locations that they read, and all are released after the third read is performed. This results in an obstruction degree of $\sum_{i=1}^{2} i = 3$. Since the write does not need to appear in the same

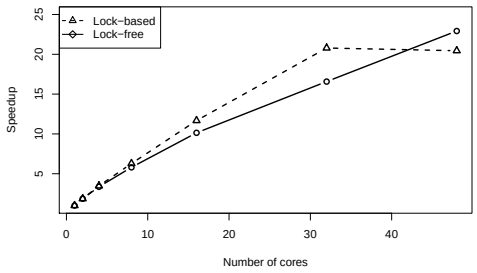


Figure 3: A lock-based and a lock-free algorithm with the same obstruction degree.

atomic unit as the reads, the final obstruction degree of both algorithms is 3. In Figure 3 we show the measured speedup of both algorithms and we observe that they provide very similar scalability, although they use different synchronization mechanisms.

In Figure 4 we show both the throughput and speedup obtained when running two versions of the same lock-free algorithm that atomically reads three memory locations and then writes one. The only difference between the two algorithms is their sequential delay, simulating local computation. The delay consists of incrementing a counter after reading the values and before revalidating them. The short algorithm increments the counter 1000 times, while the long one increments it 2000 times. We see that the throughput provided by the two algorithms is very different, as a result of the different sequential delays, but their speedup is close to identical, as a result of having the same obstruction degree. We conclude that the obstruction degree is orthogonal to the amount of local work that a process needs to perform.

6.2 A linked list

Our tests consist in running five different linked list algorithms found in the literature [19]. Each thread is executing 80% find operations, 10% insert and 10% remove on a list with 256 elements. On average operations go up to the middle of the list, so we consider the length of an operation to be 128. Figure 5 shows the plot of the obtained speedup versus the number of threads used.

The first algorithm uses a single coarse grained lock to protect the entire list. Each operation, insert, remove or find, acquires this lock before operating on the list. According to Theorem 4.5, the obstruction degree of an algorithm acquiring a coarse-grained lock is $O = n(n-1)/2$, where n is the number of objects in the system. For a list of length 256, we obtain obstruction degree 32640. We then find the concurrency efficiency to be $\mathscr{E} = 1 - \frac{32640}{256 \times 128} < \frac{1}{e}$. Having a concurrency efficiency that is smaller than $1/e$ is an artifact of computation and characterizes algorithms that do not scale above 1 processor.

The second algorithm uses the hand-over-hand locking scheme described in Section 4. Assuming that on average operations go up to the middle of the list, i.e. have length $L = n/2$, using Theorem 4.8 we find these operations to have obstruction degree $O = n/2 - 1$. This reduces to $O = 127$ for a list of length 256. We compute the concurrency efficiency of this algorithm to be $\mathscr{E} = 1 - \frac{127}{256 \times 128} \approx \frac{255}{256}$. Since $\log_{\mathscr{E}} \frac{1}{e} \approx 256$ one would expect this algorithm to scale to a large number of processors. This is however

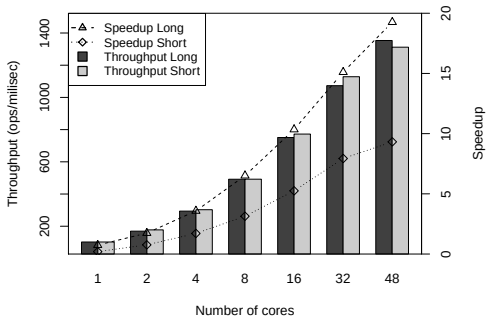


Figure 4: Algorithms with the same obstruction degree but different step complexities.

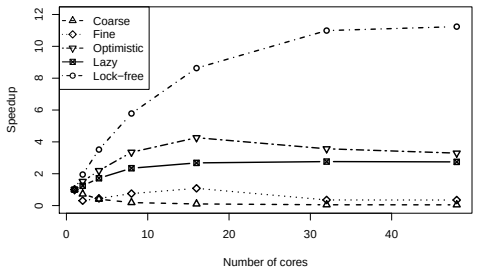


Figure 5: Comparing five linked list algorithms on a list of length 256 with 20% update operations.

not the case as the communication cost associated to acquiring such a large number of locks outweighs the cost of the object operation, which is a simple read.

The optimistic algorithm [19] operates by associating a lock with each element in the list, but each operation only acquires two locks, having obstruction degree 1. The lazy algorithm of Heller et al. [16] is also based on a fine grained locking approach. It features a find operation that does not acquire any locks, having obstruction degree 0. The insert and remove operations each acquire two locks as in the case of the optimistic algorithm, thus having obstruction degree 1. The lock-free linked list algorithm, now known as the Harris-Michael algorithm, is designed by Michael [24], who based his work on an earlier linked list algorithm of Harris [15]. No operation, insert, remove or find, acquires any locks, thus having obstruction degree 0. We approximate the concurrency efficiency of these three algorithms to $\mathscr{E} = 1 - \frac{1}{256 \times 128}$ leading to a very large number of processors that can efficiently execute it. For these algorithms the communication cost also proves to stop scalability.

Although the five algorithms do not reach the scalability limit predicted by our approach, the obstruction degree proves useful for comparing the algorithms among themselves. Thus, the coarse-grained locking approach has the highest obstruction degree and scales the worst. Next come, in this order, hand-over-hand locking, lazy, optimistic, and lock-free algorithms. Note that only the lazy and lock-free algorithms, having the lowest obstruction degree, continue to scale up to 48 cores.

6.3 STMBench7

STMBench7 [14] consists of a set of graphs and indexes designed to model CAD/CAM applications. We calculated the obstruction degree and measured the speedup for the two locking implementations provided, referred to as coarse-grained and medium-grained locking. The former uses a single read-write lock for the entire structure, while the latter uses 11 locks in total. Since read-write locks are employed by both variants, the obstruction degree of the operations was computed as described in Section 3.3.

By examining the code, we computed the obstruction degree for each of the 45 possible types of operations, which we then averaged in order to find an estimate for the obstruction degree of an operation. Thus the obstruction degree of a read-only operation is 2, when implemented using medium-grained locking, and 11, when implemented using coarse-grained locking. The average obstruction degree of an update operation using medium grained locking is $2n$, while when using coarse-grained locking it is $11n$, where n is the number of concurrent threads in the system.

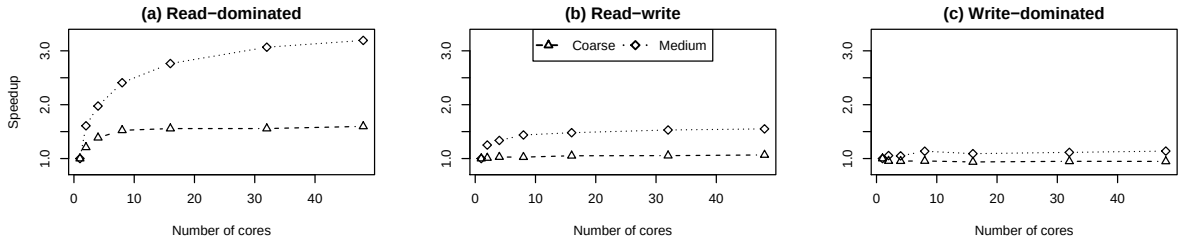


Figure 6: Performance of coarse and medium grained locking on STMBench7 with read-dominated (A), read-write (B) and write-dominated (C) workloads.

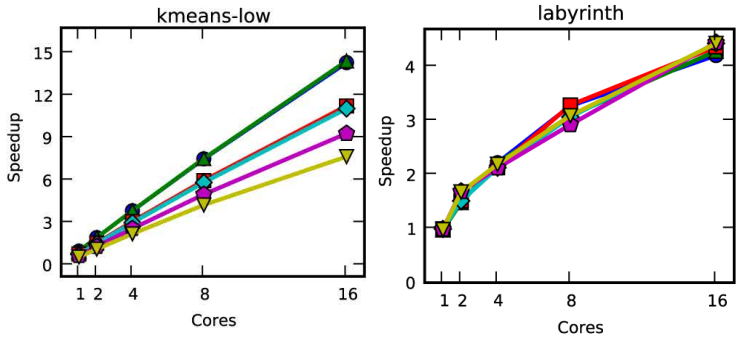


Figure 7: STAMP k-means and labyrinth benchmarks.

Figure 6-a shows the speedup obtained when executing the read-dominated workload of the benchmark, performing 90% read-only operations. The coarse-grained variant only achieves a speedup of 1.6×, whereas the medium-grained variant scales better, reaching a speedup of 3.2×. As the medium-grained version features operations having obstruction degree 2, it provides better scalability compared to the operations having obstruction degree 11 of the coarse-grained version. Since the medium-grained variant only uses 11 locks, it is also unable to scale very well. Figures 6-b and 6-c show the effects of increasing the ratio of update operations to 40% and respectively 90%. As the ratio of update operations is increased, more operations having a higher obstruction degree are executed and both the coarse and medium-grained versions scale worse. The medium-grained version scales better as its update operations have obstruction degree $2n$ compared to $11n$, which is the case of the coarse-grained version. The obstruction degree is therefore able to predict both the effect of increasing the lock granularity as well as increasing the number of the more expensive update operations.

To find the scalability limit of this benchmark, we make a rough approximation and consider the system as having 11 locks with operations of length 3 and obstruction degree 2. This represents the benchmark variant using medium-grained locking. Using these values we obtain concurrency efficiency $\mathscr{E} = 1 - \frac{2}{3\times 11} = \frac{31}{33}$ and the scalability limit $\log_{\mathscr{E}} \frac{1}{e} = 16$. Figure 6 shows that only the read-dominated workload, having operations of significantly lower obstruction degree than our initial approximation, is able to scale above 16 cores. Even with such a rough approximation that does not take into account read/write locks, we are still able to find a good estimate for the number of cores that can be efficiently used by this algorithm.

6.4 STAMP

STAMP (Stanford Transactional Applications for Multi-Processing) [25] is a suite of eight transactional memory benchmarks spanning a variety of domains from bioinformatics to machine learning. We examined the code of two benchmarks and calculated the obstruction degree of their operations.

The first, k-means, is a data mining application implementing the K-means clustering algorithm, featuring short transactions and low contention. We calculated the obstruction degree of the operations performed by the algorithm to be 150 while the length of the operations is 17 and the system size is 32000 objects. Using equation (2) we find that this algorithm could potentially scale to roughly 2500 processors. In figure 7 we reproduce the plot of the authors showing the k-means benchmark to have perfect scalability on their system with 16 processing cores. Our limit is however not tight, since communication costs will end up limiting scalability before reaching the theoretical maximum number of cores predicted by the obstruction degree.

The second, labyrinth, is an algorithm for finding paths in a three-dimensional maze, featuring long transactions and high contention. For a maze of size $32 \times 32 \times 3$, the estimated operation length is 48 and the obstruction degree 4600. Using equation (2) we find that this algorithm could potentially scale up to 30 processors. In figure 7 the plot of the authors shows the labyrinth benchmark to provide a speedup of only 4× when ran on 16 cores, making our 30-processor limit a good approximation.

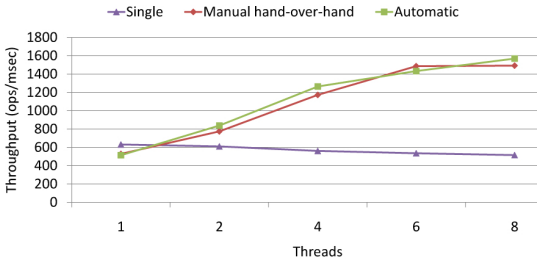


Figure 8: Throughput for a Treap data structure with a coarse-grained lock, manual hand-over-hand locking and domination locking; 70% lookups, 20% inserts and 10% removes.

6.5 Domination locking

The Domination Locking protocol of Golan-Gueta et al. [13] is used for fine-grained locking forest-like structures. Each node has a lock associated to it and operations start from the root of a tree, referred to as an exposed node. A lock on a node is released when the operation only needs to apply primitives to the subtree below that node. The authors show their locking protocol to be efficient for implementing balanced search-tree data structures, a self-adjusting heap as well as specialized tree-structures.

When executing an operation ω on a node at depth d that does not result in any rebalancing of the tree, the obstruction degree of this operation is $C(\omega_0) = d$. However, when ω needs to rebalance the tree k levels above the node at depth d, it has obstruction degree $C(\omega_k) = \frac{k^2-3k+2d}{2}$. The worst case for such an operation is when rebalancing percolates all the way to the root, thus obtaining obstruction degree $C(\omega_d) = \frac{d(d-1)}{2}$. Note that since always $k \leq d$, we have $C(\omega_k) \leq C(\omega_d)$, showing that Domination Locking is more efficient than two-phase locking.

In Figure 8 we reproduce the plot of the authors showing that domination locking scales significantly better that coarse-grained locking and just as well as a manual hand-over-hand locking implementation. According to Theorem 4.5, an operation that uses coarse-grained locking on a data structure with m nodes has obstruction degree in $O(m^2)$. As $d < m$, we have that the obstruction degree of domination locking is smaller than that of coarse-grained locking. Only in a degenerate case, when $d \approx m$ and $k \approx d$, we obtain $C(\omega_k)$ in $O(m^2)$.

The obstruction degree can thus be used to theoretically compare the locking protocol of Golan-Gueta et al. to coarse-grained locking without requiring an implementation of either one. As new locking protocols are built, our metric can be used to easily compare them agains existing ones, such as two-phase locking or Domination Locking.

7 Combining metrics

The obstruction degree can be used in conjunction with a classical complexity metric, such as the number of steps that an operation needs to take. We discuss here how the sequential step complexity conveys the performance of the system when a specified number of processes are executing, while the obstruction degree predicts the scalability of the system. Together, the two metrics provide a thorough characterization of a concurrent algorithm.

To illustrate this, consider an ordered linked list of size `n`, where the programmer wants to perform some processing on `k` of the elements. He starts off with a predefined thread-safe operation `process(t)`, where `t` is an element of the list, that he can compose in order to obtain the desired operation `processAll`. The `process` operation goes through the list in sequence starting from the head, finds element `t`, and performs the required processing. The programmer can also opt to not reuse the `process` operation but write the `processAll` operation from scratch. We consider the obstruction degree along with the sequential step complexity to reason about four possible implementations of `processAll`.

High step complexity, high obstruction degree. Here we assume the `process(t)` operation is implemented using a memory transaction and that the programmer composes several instances of this operation to obtain the `processAll`. Since the composed operation will end up starting over from the

head of the list, the step complexity is $O(nk)$. If k is in the same order as n, this complexity becomes $O(n^2)$, which is large compared to $O(n)$, which can be obtained by going through the list only once. The obstruction degree of this solution is also large because all memory locations accessed by the transaction need to be protected until the transaction commits.

Low step complexity, high obstruction degree. Assume that now the programmer wants to do the extra work of writing the processAll from scratch but, not being an expert in concurrent programming, he chooses to write it as a memory transaction. This solution has step complexity $O(n)$ since the operation needs to go through the list only once, processing elements on the way as needed. The obstruction degree is still high since all memory locations accessed by the transaction need to be protected until the transaction commits.

High step complexity, low obstruction degree. We now assume that the programmer can use a process operation implemented with a specialized transaction that only protects the element that is actually processed. He composes this operation and obtains a version of the processAll operation that restarts from the beginning of the list after processing each element but only protects the elements that must undergo the processing. The step complexity is again $O(nk)$ since processAll needs to start from the beginning of the list when processing each element but, since only the elements that need to undergo processing are protected and not all the elements of the list, this solution has a low obstruction degree.

Low step complexity, low obstruction degree. For the last case we assume that the programmer is an expert and he does not use the process method but instead he writes a highly efficient processAll from scratch. He implements it using either a very specialized transaction or more involved techniques but he ensures that it only moves through the list once and only protects the elements that need to undergo processing. The step complexity of this solution is $O(n)$ since the operation goes through the list exactly once, and the obstruction degree is also low since only the elements that need to be processed are protected.

8 Related work

Blelloch [7] describes a metric that characterizes an algorithm by summing the sequential work and the longest parallel work that need to be executed. The metric, beautiful in its simplicity, has a slightly different applicability than our obstruction degree. It is suited for applications that can be split into well determined sequential and parallel regions, while our notion of obstruction degree applies to programs using more complex synchronization.

Von Praun et al. [29] present the *dependence density*, a metric expressing the probability of having memory-level dependencies among any two randomly chosen tasks in a program phase. Computing it needs to span across tasks, while our metric can be computed locally for a single operation. Their metric is computed using runtime measurements, while calculating the obstruction degree can be done without implementing the algorithm.

The *stalls* metric [10] characterizes contention on a single object as the number of processes that have invoked a primitive on the object while not yet getting a response. However, many concurrent algorithms synchronize across several objects, a situation not covered by this metric. For instance, situations where a process holding a lock on one object applies a primitive on another object fall outside of the scope of this metric. Attiya et al. count the number of stalls that an operation experiences in the worst [2] as well as the best [3] possible execution when requiring operations to be linearizable [20]. Since in our model operations can release protection elements before finishing, linearizability is no longer required. Unlike the stalls metric, the obstruction degree also applies to locking algorithms, which make up a large part of existing concurrent programs.

Afek et al. [1] characterize an operation as having *d-local step complexity* if the number of steps performed by the operation in a given execution interval is bounded by a function of the number of primitives applied within distance d in the conflict graph of the given interval. They define an algorithm as having *d-local contention* if two operations access the same object only if the distance between them in the conflict graph of their joint execution interval is at most d. Like the number of stalls, this metric

does not apply to locking algorithms and can be computed only by considering an execution of the entire system.

Ellen et al. [12] introduce the *obstruction-free step complexity* as the maximum number of steps that an operation needs to perform if no other processes are taking steps. This measure considers the favorable case when other processes are not taking steps, whereas the obstruction degree considers the average case, where other processes are taking steps that may or may not interfere with the operation being considered. Furthermore, this metric, unlike the obstruction degree, does not apply to lock-based programs, and can only be calculated when considering a complete execution of the system, not just one operation in isolation.

More involved techniques, such as *multiple linear regression* [8] and *artificial neural networks* [21], rely on information such as hardware performance counters obtained from a few runs of the program when making predictions. Singh et al. [28] compared these two approaches on different workloads and concluded that both make accurate predictions regarding system scalability. Our notion of obstruction degree complements these techniques. Programmers can use the obstruction degree in the algorithm design phase for choosing the most promising algorithms, which can then be implemented and compared using such more involved techniques.

9 Limitations

Our notion of obstruction degree is aimed to be a simple way to approximate a priori the scalability of concurrent algorithms obtained by adding synchronization to sequential algorithms. Not surprisingly, the simplicity comes with some limitations. As we pointed out in the introduction, predictions made using the obstruction degree are very accurate when any object in the system is as likely to be accessed next by any operation. With hotspots, the accuracy of the predictions gradually degrades. (Still, as we show in Section 6, our metric is able to make reasonably good predictions even in situations where objects are not accessed in a uniform fashion.)

One can also imagine cases where the algorithm has so much structure that it "tricks" the obstruction degree. If the semantics of all the operations in the system are known to the programmer, program structure can also be used to obstruct other processes. If $o, o' \in \mathcal{O}$ are two objects in the system, and the only way to access o' is by obtaining the response from a primitive applied to o, a process holding the protection element $\epsilon(o)$ will indirectly obstruct other processes from accessing o', even if $\epsilon(o) \neq \epsilon(o')$. To illustrate this, consider a linked list where each element is protected by a distinct lock. Each operation, insert, remove or find, instead of acquiring locks using the hand-over-hand technique [6], only acquires the lock corresponding to the first element. This lock is then kept until the process finishes executing the operation. This method does not allow two processes to access the list concurrently since while one process holds a lock on the first element, no other process can access any element of the list. The obstruction degree of this algorithm is 0, indicating very good scalability. However, such an algorithm does not allow any two operations to execute concurrently and does not in fact scale.

To adapt the obstruction degree to such a situation, for any object o which a process must access in order to obtain access to k other objects, one must replace the acquisition event $\langle a(\epsilon(o)), p\rangle$ by the sequence of acquisition events $\langle a(\epsilon(o)^{(1)}), p\rangle, \ldots, \langle a(\epsilon(o)^{(k)}), p\rangle$ and similarly for the release event. This technique gives operations on the above linked list an obstruction degree in $O(n^2)$, which is identical to that of a coarse-grained lock. This is accurate since, like when using a coarse-grained lock, only a single operation can execute at any given time.

Another situation where the obstruction degree could make false predictions regarding system scalability is when certain objects or protection elements have a higher probability of being accessed than others. On the one hand, an operation could have a high obstruction degree, but if the probability of another process needing to acquire the same protection elements is low, the algorithm will scale well. On the other hand, an operation could have a low obstruction degree, but could acquire protection elements that have a high probability of being needed by other operations, thus significantly hampering scalability.

To overcome this, the algorithm designer can use knowledge about the algorithm to assign costs to protection elements. A protection element $\epsilon(o)$ has a higher cost than another protection element $\epsilon(o')$ if $\epsilon(o)$ is acquired by more operations than $\epsilon(o')$. Then if the cost of protection element $\epsilon(o)$ is $cost(\epsilon(o))$, we compute the cost of the protected set of an operation ω to be $cost(P(\omega)) = \sum_{\epsilon(o)\in P(\omega)} cost(\epsilon(o))$. We also

change rule **OD2** of Definition 3.1 to be **OD2':** if $H = H' \cdot \langle a(\epsilon(o)), p \rangle$, then $C(H) = C(H') + cost(P(H'))$. Using this technique we can make better predictions for algorithms featuring "hotspots", protection elements that are very likely to be acquired, or protection elements that have a very small probability of being acquired.

Finally, it is important to recall that our metric is intended as an abstract cost model, simple enough to be used by a wide class of users, yet general enough to make useful comparisons independently of specific hardware. It does not take into account factors such as specific optimizations or the effects of specific hardware. When a more precise and hardware dependent comparison is needed, our notion of obstruction degree is useful as a first approximation, allowing the best algorithms to be selected and later compared using more involved techniques.

References

[1] Yehuda Afek, Michael Merritt, Gadi Taubenfeld, and Dan Touitou. Disentangling multi-object operations. In *PODC*, 1997.

[2] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kouznetsov. Synchronizing without locks is inherently expensive. In *PODC*, 2006.

[3] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 2009.

[4] Hagit Attiya and Eshcar Hillel. Highly-concurrent multi-word synchronization. In *ICDCN*, 2008.

[5] Hagit Attiya and Welch Jennifer. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. John Wiley & Sons, 2nd edition, 2004.

[6] Rudolf Bayer and Schkolnick Mario. Concurrency of operations on b-trees. *Acta Inf.*, 1977.

[7] Guy E. Blelloch. Parallel thinking. In *PPoPP*, 2009.

[8] Matthew Curtis-Maury, Filip Blagojevic, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE TPDS*, 2008.

[9] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *PLDI*, 2009.

[10] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *J. ACM*, 1997.

[11] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *C. ACM*, 1976.

[12] Faith Ellen Fich, Victor Luchangco, Mark Moir, and Nir Shavit. Obstruction-free step complexity: lock-free DCAS as an example. In *DISC*, 2005.

[13] Guy Golan-Gueta, Nathan Bronson, Alex Aiken, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Automatic fine-grain locking using shape properties. In *OOPSLA*, 2011.

[14] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. In *EuroSys*, 2007.

[15] Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, 2001.

[16] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer III, and Nir Shavit. A lazy concurrent list-based set algorithm. *PPL*, 2007.

[17] Maurice Herlihy. Wait-free synchronization. *ACM TOPLAS*, 1991.

[18] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *ISCA*, 1993.

[19] Maurice Herlihy and Nir Shavit. *The art of multiprocessor programming.* Morgan Kaufmann, 2008.

[20] Maurice Herlihy and Jeannette Wing. Linearizability: a correctness condition for concurrent objects. *ACM TOPLAS*, 1990.

[21] Engin Ipek, Sally A. McKee, Karan Singh, Rich Caruana, Bronis R. de Supinski, and Martin Schulz. Efficient architectural design space exploration via predictive modeling. *ACM TACO*, 2008.

[22] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *PODC*, 1994.

[23] E. H. Mckinney. Generalized birthday problem. *Amer. Math. Monthly*, 1966.

[24] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, 2002.

[25] Chi Cao Minh, Jaewoong Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, 2008.

[26] Torvald Riegel, Pascal Felber, and Christof Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

[27] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms.* Addison-Wesley, 2013.

[28] Karan Singh, Matthew Curtis-Maury, Sally A. McKee, Filip Blagojević, Dimitrios S. Nikolopoulos, Bronis R. de Supinski, and Martin Schulz. Comparing scalability prediction strategies on an SMP of CMPs. In *EuroPar*, 2010.

[29] Christoph von Praun, Rajesh Bordawekar, and Calin Cascaval. Modeling optimistic concurrency using quantitative dependence analysis. In *PPoPP*, 2008.